Name of Student: Pushkar Sane			
Roll Number: 45		**Lab Assignment Number: 6**	
Title of Lab Assignment: To implement programs on Data Structures using Python.			
DOP:		**DOS:**	
CO Mapped: **CO3**	**PO Mapped:** **PO5, PSO1**	**Signature:**	**Marks:**

Practical No. 6

Aim: To implement programs on data structure using python.

1. To Create, Traverse, Insert and remove data using Linked List
2. Implementation of stacks
3. Implementation of Queue
4. Implementation of Dequeue

Description:

A data structure is a fundamental concept in computer science and programming. It serves as an organizational framework for efficiently storing, accessing, and manipulating data. Data structures come in various forms, including arrays, linked lists, stacks, queues, trees, graphs, and hash tables. Each data structure has specific characteristics and is chosen based on the problem's requirements. Data structures play a crucial role in optimizing algorithm design and the development of software applications by providing efficient data management and retrieval mechanisms. Understanding data structures is essential for building efficient and organized computer programs.

1. **Linked List:**
   A linked list is a linear data structure consisting of a sequence of elements, each connected to the next element through pointers. Here's a detailed overview of creating, traversing, inserting, and removing data in a linked list:
   - **Creation:** To create a linked list, you start with a head node, which is the first element in the list. Each node in the list contains two components: data and a reference (or pointer) to the next node in the sequence.
   - **Traversing:** Traversing a linked list involves moving through the list one node at a time. You start at the head and follow the references to each subsequent node until you reach the end (usually when the next reference is null). This allows you to access and display the data in each node.
   - **Insertion:** Inserting data in a linked list typically involves adding a new node to the list. To insert data at a specific position, you adjust the references to link the new node correctly within the list.

- **Removal:** Removing data from a linked list requires finding the node to be removed and updating the references to bypass that node. The node can then be deleted.

2. **Stack:**
   A stack is a linear data structure that follows the Last-In-First-Out (LIFO) principle. It's commonly used for managing function calls, tracking execution history, and solving problems recursively. Here's a detailed overview of stack operations:
   - **Push:** The push operation adds an element to the top of the stack.
   - **Pop:** The pop operation removes and returns the top element from the stack.
   - **Peek:** The peek operation allows you to view the top element without removing it.
   - **Empty:** A check to see if the stack is empty.
   - **Size:** Determining the number of elements in the stack.

3. **Queue:**
   A queue is another linear data structure, but it follows the First-In-First-Out (FIFO) principle. It's used in scenarios where tasks should be processed in the order they are received. Here's a detailed overview of queue operations:
   - **Enqueue:** The enqueue operation adds an element to the rear (or end) of the queue.
   - **Dequeue:** The dequeue operation removes and returns the element at the front of the queue.
   - **Front:** Viewing the element at the front of the queue without removing it.
   - **Rear:** Viewing the element at the rear of the queue without removing it.
   - **Empty:** Checking if the queue is empty.
   - **Size:** Determining the number of elements in the queue.

4. **Double-Ended Queue (Deque):**
   A double-ended queue, or deque, is a versatile data structure that allows elements to be added or removed from both ends. It's useful in scenarios where data needs to be efficiently accessed from either end. Here's a detailed overview of deque operations:
   - **Enqueue Front:** Adding an element to the front of the deque.
   - **Enqueue Rear:** Adding an element to the rear of the deque.
   - **Dequeue Front**: Removing and returning the element at the front of the deque.

- **Dequeue Rear:** Removing and returning the element at the rear of the deque.
- **Front:** Viewing the element at the front of the deque without removing it.
- **Rear:** Viewing the element at the rear of the deque without removing it.
- **Empty:** Checking if the deque is empty.
- **Size:** Determining the number of elements in the deque.

These data structures and their associated operations are fundamental building blocks in computer science and software development. Understanding how to create, manipulate, and applying them is essential for solving a wide range of problems efficiently and effectively.

1. **To Create, Traverse, Insert and remove data using Linked List.**

Code:

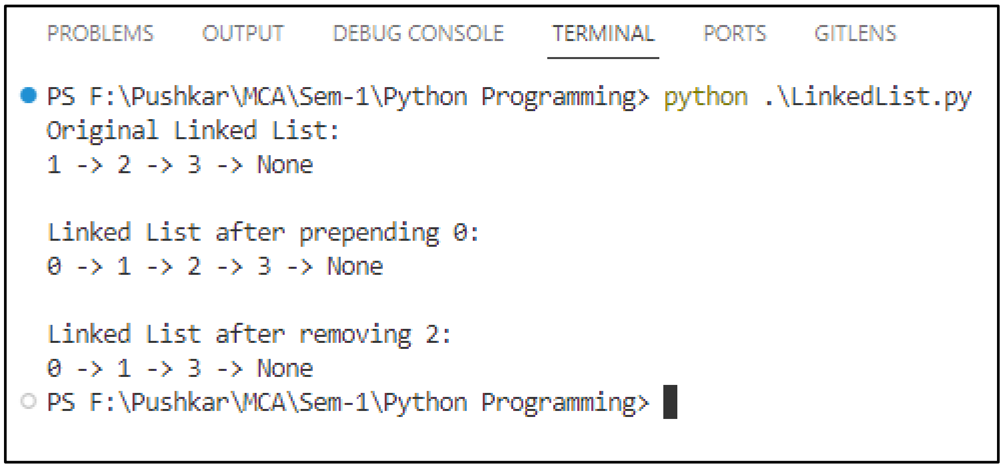
```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None

    # Insert at the end of the linked list
    def append(self, data):
        new_node = Node(data)
        if not self.head:
            self.head = new_node
            return
        current = self.head
        while current.next:
            current = current.next
        current.next = new_node

    # Traverse and print the linked list
    def display(self):
        current = self.head
        while current:
            print(current.data, end=" -> ")
            current = current.next
        print("None")

    # Insert data at the beginning of the linked list
    def prepend(self, data):
        new_node = Node(data)
        new_node.next = self.head
```

```
            self.head = new_node

        # Remove a specific data element from the linked list
        def remove(self, data):
            if not self.head:
                return
            if self.head.data == data:
                self.head = self.head.next
                return
            current = self.head
            while current.next:
                if current.next.data == data:
                    current.next = current.next.next
                    return
                current = current.next

# Example usage
linked_list = LinkedList()
linked_list.append(1)
linked_list.append(2)
linked_list.append(3)

print("Original Linked List:")
linked_list.display()

linked_list.prepend(0)
print("\nLinked List after prepending 0:")
linked_list.display()

linked_list.remove(2)
print("\nLinked List after removing 2:")
linked_list.display()
```

Conclusion:

In conclusion, the Python program presented a basic implementation of a singly linked list, including operations to create, traverse, insert, remove, find the length, search for elements, insert at specific positions, reverse the list, and retrieve elements from the end. This program serves as a practical foundation for understanding and working with linked lists, an essential data structure in computer science and programming.

Output:



```
PS F:\Pushkar\MCA\Sem-1\Python Programming> python .\LinkedList.py
Original Linked List:
1 -> 2 -> 3 -> None

Linked List after prepending 0:
0 -> 1 -> 2 -> 3 -> None

Linked List after removing 2:
0 -> 1 -> 3 -> None
PS F:\Pushkar\MCA\Sem-1\Python Programming>
```

2. **Implementation of stack.**

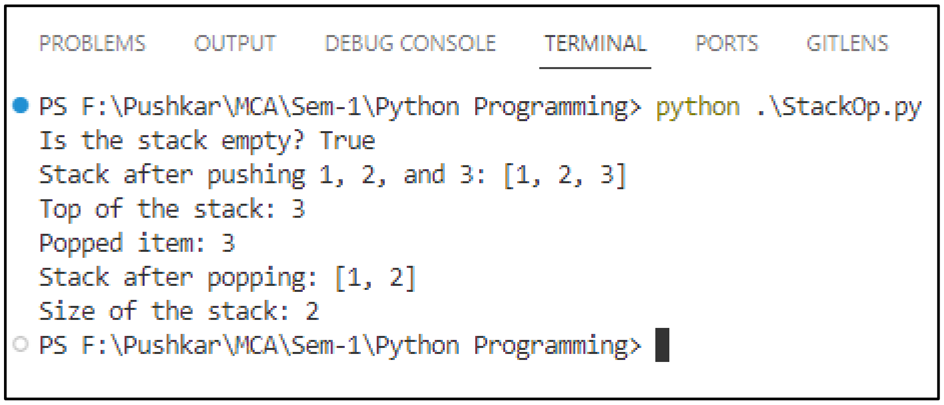
Code:

```
class Stack:
    def __init__(self):
        self.items = []

    def is_empty(self):
        return len(self.items) == 0

    def push(self, item):
        self.items.append(item)

    def pop(self):
        if not self.is_empty():
            return self.items.pop()
        else:
            return "Stack is empty"

    def peek(self):
        if not self.is_empty():
            return self.items[-1]
        else:
            return "Stack is empty"

    def size(self):
        return len(self.items)

# Example usage
stack = Stack()
print("Is the stack empty?", stack.is_empty())
stack.push(1)
stack.push(2)
stack.push(3)

print("Stack after pushing 1, 2, and 3:", stack.items)
print("Top of the stack:", stack.peek())
popped_item = stack.pop()
print(f"Popped item: {popped_item}")
print("Stack after popping:", stack.items)
print("Size of the stack:", stack.size())
```

Conclusion:

The Python stack implementation showcased basic stack operations following the Last-In-First-Out (LIFO) principle. Stacks are versatile for tasks like managing function calls and recursion. The code demonstrated pushing, popping, peeking, and checking stack properties, providing fundamental tools for data management and algorithm design.

Output:



```
PS F:\Pushkar\MCA\Sem-1\Python Programming> python .\StackOp.py
Is the stack empty? True
Stack after pushing 1, 2, and 3: [1, 2, 3]
Top of the stack: 3
Popped item: 3
Stack after popping: [1, 2]
Size of the stack: 2
PS F:\Pushkar\MCA\Sem-1\Python Programming>
```

3. **Implementation of Queue.**

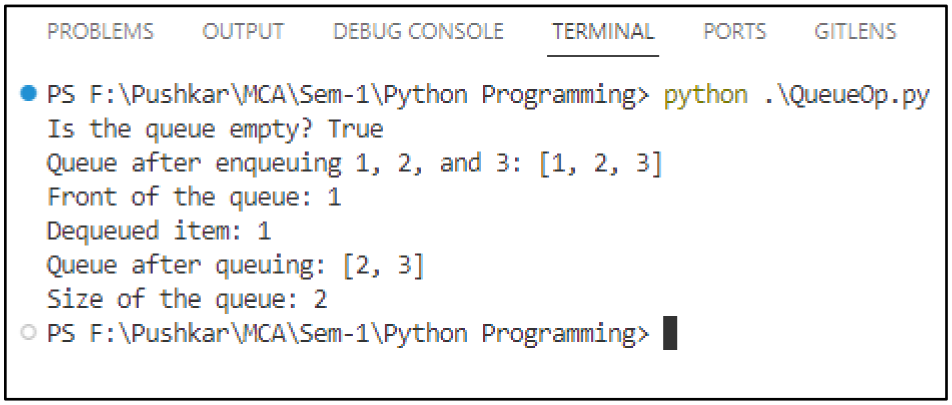
Code:

```
class Queue:
    def __init__(self):
        self.items = []

    def is_empty(self):
        return len(self.items) == 0

    def enqueue(self, item):
        self.items.append(item)

    def dequeue(self):
        if not self.is_empty():
            return self.items.pop(0)
        else:
            return "Queue is empty"

    def peek(self):
        if not self.is_empty():
            return self.items[0]
        else:
            return "Queue is empty"
    def size(self):
        return len(self.items)

# Example usage
queue = Queue()
print("Is the queue empty?", queue.is_empty())
queue.enqueue(1)
queue.enqueue(2)
queue.enqueue(3)
print("Queue after enqueuing 1, 2, and 3:", queue.items)
print("Front of the queue:", queue.peek())
dequeued_item = queue.dequeue()
print(f"Dequeued item: {dequeued_item}")
print("Queue after queuing:", queue.items)
print("Size of the queue:", queue.size())
```

Conclusion:

The Python queue implementation embodies First-In-First-Out (FIFO) behavior, essential for orderly data processing. It showcases basic operations- enqueue, dequeue, peek, and property checks- vital for tasks like task scheduling and data buffering. Queue proficiency is invaluable in software development and computer science.

Output:



```
PS F:\Pushkar\MCA\Sem-1\Python Programming> python .\QueueOp.py
Is the queue empty? True
Queue after enqueuing 1, 2, and 3: [1, 2, 3]
Front of the queue: 1
Dequeued item: 1
Queue after queuing: [2, 3]
Size of the queue: 2
PS F:\Pushkar\MCA\Sem-1\Python Programming>
```

4. **Implementation of Dequeue.**

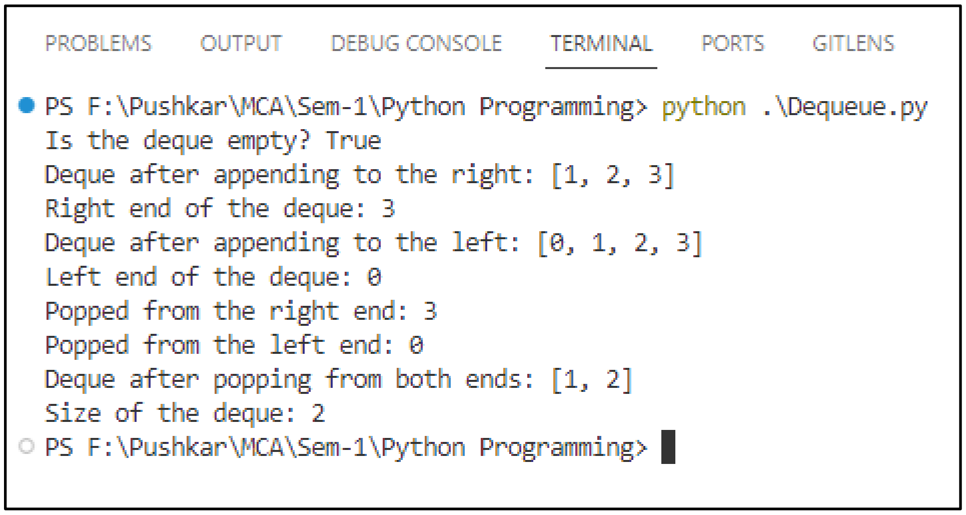
Code:

```
class Deque:
    def __init__(self):
        self.items = []

    def is_empty(self):
        return len(self.items) == 0

    def append_right(self, item):
        self.items.append(item)

    def append_left(self, item):
        self.items.insert(0, item)

    def pop_right(self):
        if not self.is_empty():
            return self.items.pop()
        else:
            return "Deque is empty"

    def pop_left(self):
        if not self.is_empty():
            return self.items.pop(0)
        else:
            return "Deque is empty"

    def peek_right(self):
        if not self.is_empty():
            return self.items[-1]
        else:
            return "Deque is empty"
    def peek_left(self):
        if not self.is_empty():
            return self.items[0]
        else:
            return "Deque is empty"

    def size(self):
        return len(self.items)

# Example usage
deque = Deque()
print("Is the deque empty?", deque.is_empty())
deque.append_right(1)
deque.append_right(2)
deque.append_right(3)
print("Deque after appending to the right:", deque.items)
print("Right end of the deque:", deque.peek_right())
deque.append_left(0)
print("Deque after appending to the left:", deque.items)
print("Left end of the deque:", deque.peek_left())
popped_right = deque.pop_right()
print(f"Popped from the right end: {popped_right}")
popped_left = deque.pop_left()
print(f"Popped from the left end: {popped_left}")
print("Deque after popping from both ends:", deque.items)
print("Size of the deque:", deque.size())
```

Conclusion:

In this Python implementation, we introduced a double-ended queue (deque) class with basic operations for efficient element addition and removal from both ends. Deques are versatile data structures useful in various scenarios, such as managing data with distinct priorities and processing elements in an ordered manner. Understanding deque operations is valuable for solving problems in computer science and software development.

<u>Output:</u>



```
PS F:\Pushkar\MCA\Sem-1\Python Programming> python .\Dequeue.py
Is the deque empty? True
Deque after appending to the right: [1, 2, 3]
Right end of the deque: 3
Deque after appending to the left: [0, 1, 2, 3]
Left end of the deque: 0
Popped from the right end: 3
Popped from the left end: 0
Deque after popping from both ends: [1, 2]
Size of the deque: 2
PS F:\Pushkar\MCA\Sem-1\Python Programming>
```